
XWorkflows Documentation

Release 1.1.0

Raphaël Barrois

Apr 29, 2021

Contents

1	Getting started	3
1.1	Declaring workflows	3
1.2	Applying a workflow	4
1.3	Using the transitions	4
1.4	Custom transition code	4
1.5	Hooks	5
2	Contents	7
2.1	Reference	7
2.2	Internals	14
2.3	ChangeLog	26
3	Resources	29
4	Indices and tables	31
	Python Module Index	33
	Index	35

XWorkflows is a library designed to bring a simple approach to workflows in Python.

It provides:

- Simple workflow definition
- Running code when performing transitions
- Hooks for running extra code before/after the transition
- A hook for logging performed transitions

You can also refer to the [django_xworkflows](#) project for integration with Django.

First, install the `xworkflows` package:

```
pip install xworkflows
```

1.1 Declaring workflows

You can now define a *Workflow*:

```
import xworkflows

class MyWorkflow(xworkflows.Workflow):
    states = (
        ('init', "Initial state"),
        ('ready', "Ready"),
        ('active', "Active"),
        ('done', "Done"),
        ('cancelled', "Cancelled"),
    )

    transitions = (
        ('prepare', 'init', 'ready'),
        ('activate', 'ready', 'active'),
        ('complete', 'active', 'done'),
        ('cancelled', ('ready', 'active'), 'cancelled'),
    )

    initial_state = 'init'
```

1.2 Applying a workflow

In order to apply that workflow to an object, you must:

- Inherit from `xworkflows.WorkflowEnabled`
- Define one (or more) class attributes as `Workflow` instances.

Here is an example:

```
class MyObject(xworkflows.WorkflowEnabled):  
    state = MyWorkflow()
```

1.3 Using the transitions

With the previous definition, some methods have been *magically* added to your object definition (have a look at `WorkflowEnabledMeta` to see how).

There is now one method per transition defined in the workflow:

```
>>> obj = MyObject()  
>>> obj.state  
<StateWrapper: <State: 'init'>>  
>>> obj.state.name  
'init'  
>>> obj.state.title  
'Initial state'  
>>> obj.prepare()  
>>> obj.state  
<StateWrapper: <State: 'ready'>>  
>>> obj.state.name  
'ready'  
>>> obj.state.title  
'Ready'
```

As seen in the example above, calling a transition automatically updates the state of the workflow.

Only transitions compatible with the current state may be called:

```
>>> obj.state  
<StateWrapper: <State: 'ready'>>  
>>> obj.complete()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
InvalidTransitionError: Transition 'complete' isn't available from state 'ready'.
```

1.4 Custom transition code

It is possible to define explicit code for a transition:

```
class MyObject(xworkflows.WorkflowEnabled):  
    state = MyWorkflow()  
  
    @xworkflows.transition()
```

(continues on next page)

(continued from previous page)

```
def activate(self, user):
    self.activated_by = user
    print("State is %s" % self.state.name)

obj = MyObject()
```

When calling the transition, the custom code is called before updating the state:

```
>>> obj.state
<StateWrapper: <State: 'init'>>
>>> obj.prepare()
>>> obj.state
<StateWrapper: <State: 'ready'>>
>>> obj.activate('blah')
State is ready
>>> obj.state
<StateWrapper: <State: 'active'>>
>>> obj.activated_by
'blah'
```

1.5 Hooks

Other functions can be hooked onto transitions, through the *before_transition()*, *after_transition()*, *transition_check()*, *on_enter_state()* and *on_leave_state()* decorators:

```
class MyObject(xworkflows.WorkflowEnabled):
    state = MyWorkflow()

    @xworkflows.before_transition('foobar', 'gobaz')
    def hook(self, *args, **kwargs):
        pass
```


2.1 Reference

The XWorkflow library has two main aspects:

- Defining a workflow;
- Using a workflow on an object.

2.1.1 Defining a workflow

A workflow is defined by subclassing the *Workflow* class, and setting a few specific attributes:

```
class MyWorkflow(xworkflows.Workflow):

    # The states in the workflow
    states = (
        ('init', _(u"Initial state")),
        ('ready', _(u"Ready")),
        ('active', _(u"Active")),
        ('done', _(u"Done")),
        ('cancelled', _(u"Cancelled")),
    )

    # The transitions between those states
    transitions = (
        ('prepare', 'init', 'ready'),
        ('activate', 'ready', 'active'),
        ('complete', 'active', 'done'),
        ('cancel', ('ready', 'active'), 'cancelled'),
    )

    # The initial state of objects using that workflow
    initial_state = 'init'
```

Those attributes will be transformed into similar attributes with friendlier APIs:

- `states` is defined as a list of two-tuples and converted into a `StateList`
- `transitions` is defined as a list of three-tuples and converted into a `TransitionList`
- `initial_state` is defined as the `name` of the initial `State` of the `Workflow` and converted into the appropriate `State`

Accessing Workflow states and transitions

The `states` attribute, a `StateList` instance, provides a mixed dictionary/object API:

```
>>> MyWorkflow.states.init
State('init')
>>> MyWorkflow.states.init.title
u"Initial state"
>>> MyWorkflow.states['ready']
State('ready')
>>> 'active' in MyWorkflow.states
True
>>> MyWorkflow.states.init in MyWorkflow.states
True
>>> list(MyWorkflow.states) # definition order is kept
[State('init'), State('ready'), State('active'), State('done'), State('cancelled')]
```

The `transitions` attribute of a `Workflow` is a `TransitionList` instance, exposing a mixed dictionary/object API:

```
>>> MyWorkflow.transitions.prepare
Transition('prepare', [State('init')], State('ready'))
>>> MyWorkflow.transitions['cancel']
Transition('cancel', [State('ready'), State('active')], State('cancelled'))
>>> 'activate' in MyWorkflow.transitions
True
>>> MyWorkflow.transitions.available_from(MyWorkflow.states.ready)
[Transition('activate'), Transition('cancel')]
>>> list(MyWorkflow.transitions) # Definition order is kept
[Transition('prepare'), Transition('activate'), Transition('complete'), Transition(
  ↳ 'cancel')]
```

2.1.2 Using a workflow

The process to apply a `Workflow` to an object is quite straightforward:

- Inherit from `WorkflowEnabled`
- Define one or more class-level attributes as `foo = SomeWorkflow()`

These attributes will be transformed into `StateProperty` objects, acting as a wrapper around the `State` held in the object's internal `__dict__`.

For each transition of each related `Workflow`, the `WorkflowEnabledMeta` metaclass will add or enhance a method for each transition, according to the following rules:

- If a class method is decorated with `transition('XXX')` where XXX is the name of a transition, that method becomes the `ImplementationWrapper` for that transition

- For each remaining transition, if a method exists with the same name *and* is decorated with the `transition()` decorator, it will be used for the `ImplementationWrapper` of the transition. Methods with a transition name but no decorator will raise a `TypeError` – this ensures that all magic is somewhat explicit.
- For all transitions which didn't have an implementation in the class definition, a new method is added to the class definition. They have the same name as the transition, and a `noop()` implementation. `TypeError` is raised if a non-callable attribute already exists for a transition name.

Accessing the current state

For a `WorkflowEnabled` object, each `<attr> = SomeWorkflow()` definition is translated into a `StateProperty` object, which adds a few functions to a plain attribute:

- It checks that any value set is a valid `State` from the related `Workflow`:

```
>>> obj = MyObject()
>>> obj.state = State('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Value State('foo') is not a valid state for workflow MyWorkflow.
```

- It defaults to the `initial_state` of the `Workflow` if no value was set:

```
>>> obj = MyObject()
>>> obj.state
State('init')
```

- It wraps retrieved values into a `StateWrapper`, which adds a few extra attributes:

- Access to the related workflow:

```
>>> obj.state.workflow
<Workflow: MyWorkflow>
```

- List of accessible transitions:

```
>>> obj.state.transitions
[Transition('accept')]
```

- Easy testing of the current value:

```
>>> obj.state.is_init
True
>>> obj.state.is_ready
False
```

- Native equivalence to the `state's name`:

```
>>> obj.state == 'init'
True
>>> obj.state == 'ready'
False
>>> obj.state in ['init', 'ready']
True
```

Note: This behavior should only be used when accessing the `State` objects from the `Workflow.states` list is impossible, e.g comparison with external data (URL, database, ...).

Using `State` objects or the `is_XXX` attributes protects from typos in the code (`AttributeError` would be raised), whereas raw strings provide no such guarantee.

- Easily setting the current value:

```
>>> obj.state = MyWorkflow.states.ready
>>> obj.state.is_ready
True

>>> # Setting from a state name is also possible
>>> obj.state = 'ready'
>>> obj.state.is_ready
True
```

Note: Setting the state without going through transitions defeats the goal of xworkflows; this feature should only be used for faster testing or when saving/restoring objects from external storage.

2.1.3 Using transitions

Defining a transition implementation

In order to link a state change with specific code, a `WorkflowEnabled` object must simply have a method decorated with the `transition()` decorator.

If that method cannot be defined with the name of the related `Transition`, the name of that `Transition` should be passed as first argument to the `transition()` decorator:

```
class MyObject(xworkflows.WorkflowEnabled):

    state = MyWorkflow()

    @xworkflows.transition()
    def accept(self):
        pass

    @xworkflows.transition('cancel')
    def do_cancel(self):
        pass
```

Once decorated, any call to that method will perform the following steps:

1. Check that the current `State` of the object is a valid source for the target `Transition` (raises `InvalidTransitionError` otherwise);
2. Checks that all optional `transition_check()` hooks, if defined, returns `True` (raises `ForbiddenTransition` otherwise);
3. Run optional `before_transition()` and `on_leave_state()` hooks
4. Call the code of the function;
5. Change the `State` of the object;
6. Call the `Workflow.log_transition()` method of the related `Workflow`;
7. Run the optional `after_transition()` and `on_enter_state()` hooks, if defined.

Transitions for which no implementation was defined will have a basic `noop()` implementation.

Controlling transitions

According to the order above, preventing a `State` change can be done:

- By returning `False` in a custom `transition_check()` hook;
- By raising any exception in a custom `before_transition()` or `on_leave_state()` hook;
- By raising any exception in the actual implementation.

Hooks

Additional control over the transition implementation can be obtained via hooks. 5 kinds of hooks exist:

- `transition_check()`: those hooks are called just after the `State` check, and should return `True` if the transition can proceed. No argument is provided to the hook.
- `before_transition()`: hooks to call just before running the actual implementation. They receive the same `*args` and `**kwargs` as passed to the actual implementation (but can't modify them).
- `after_transition()`: those hooks are called just after the `State` has been updated. It receives:
 - `res`: the return value of the actual implementation;
 - `*args` and `**kwargs`: the arguments passed to the actual implementation
- `on_leave_state()`: functions to call just before leaving a state, along with the `before_transition()` hooks. They receive the same arguments as a `before_transition()` hook.
- `on_enter_state()`: hooks to call just after entering a new state, along with `after_transition()` hooks. They receive the same arguments as a `after_transition()` hook.

The hook decorators all accept the following arguments:

- A list of `Transition` names (for transition-related hooks) or `State` names (for state-related hooks); if empty, the hook will apply to all transitions:

```
@xworkflows.before_transition()
@xworkflows.after_transition('foo', 'bar')
def hook(self, *args, **kwargs):
    pass
```

- As a keyword `field=` argument, the name of the field whose transitions the hook applies to (when an instance uses more than one workflow):

```
class MyObject(xworkflows.WorkflowEnabled):
    state1 = SomeWorkflow()
    state2 = AnotherWorkflow()

    @xworkflows.on_enter_state(field='state2')
    def hook(self, res, *args, **kwargs):
        # Only called for transitions on state2.
        pass
```

- As a keyword `priority=` argument (default: 0), the priority of the hook; hooks are applied in decreasing priority order:

```
class MyObject(xworkflows.WorkflowEnabled):
    state = SomeWorkflow()

    @xworkflows.before_transition('*', priority=-1)
    def last_hook(self, *args, **kwargs):
        # Will be called last
        pass

    @xworkflows.before_transition('foo', priority=10)
    def first_hook(self, *args, **kwargs):
        # Will be called first
        pass
```

Hook decorators can also be stacked, in order to express complex hooking systems:

```
@xworkflows.before_transition('foobar', priority=4)
@xworkflows.on_leave_state('baz')
def hook(self, *args, **kwargs):
    pass
```

Hook call order

The order in which hooks are applied is computed based on the following rules:

- **Build the list of hooks to apply**
 - When testing if a transition can be applied, use all `transition_check()` hooks
 - Before performing a transition, use all `before_transition()` and `on_leave_state()` hooks
 - After performing a transition, use all `after_transition()` and `on_enter_state()` hooks
- Sort that list from higher to lower priority, and in alphabetical order if priority match

In the following code snippet, the order is hook3, hook1, hook4, hook2:

```
@xworkflows.before_transition()
def hook1(self):
    pass

@xworkflows.before_transition(priority=-1)
def hook2(self):
    pass

@xworkflows.before_transition(priority=10)
def hook3(self):
    pass

@xworkflows.on_leave_state()
def hook4(self):
    pass
```

Old-style hooks

Hooks can also be bound to the implementation at the `transition()` level:

```
@xworkflows.transition(check=some_fun, before=other_fun, after=something_else)
def accept(self):
    pass
```

Deprecated since version 0.4.0: Use `before_transition()`, `after_transition()` and `transition_check()` instead; will be removed in 0.5.0.

The old behaviour did not allow for hook overriding in inherited workflows.

Checking transition availability

Some programs may need to display *available* transitions, without calling them. Instead of checking manually the *state* of the object and calling the appropriate `transition_check()` hooks if defined, you should simply call `myobj.some_transition.is_available()`:

```
class MyObject(WorkflowEnabled):
    state = MyWorkflow
    x = 13

    @transition_check('accept')
    def check(self):
        return self.x == 42

    def accept(self):
        pass

    @transition()
    def cancel(self):
        pass
```

```
>>> obj = MyObject()
>>> obj.accept.is_available() # Forbidden by 'check'
False
>>> obj.cancel.is_available() # Forbidden by current state
False
>>> obj.x = 42
>>> obj.accept.is_available()
True
```

Logging transitions

The `log_transition()` method of a *Workflow* allows logging each *Transition* performed by an object using that *Workflow*.

This method is called with the following arguments:

- `transition`: the *Transition* just performed
- `from_state`: the *State* in which the object was just before the transition
- `instance`: the object to which the transition was applied
- `*args`: the arguments passed to the transition implementation
- `**kwargs`: the keyword arguments passed to the transition implementation

The default implementation logs (with the `logging` module) to the `xworkflows.transitions` logger.

This behaviour can be overridden on a per-workflow basis: simply override the `Workflow.log_transition()` method.

Advanced customization

In order to perform advanced tasks when running transitions, libraries may hook directly at the `ImplementationWrapper` level.

For this, custom `Workflow` classes should override the `Workflow.implementation_class` attribute with their custom subclass and add extra behaviour there.

Possible customizations would be:

- Wrapping implementation call and state update in a database transaction
- Persisting the updated object after the transition
- Adding workflow-level hooks to run before/after the transition
- Performing the same sanity checks for all objects using that `Workflow`

2.2 Internals

This document presents the various classes and components of XWorkflows.

Note: All objects defined in the `base` module should be considered internal API and subject to change without notice.

Public API consists of the public methods and attributes of the following objects:

- The `transition()` function;
 - The `before_transition()`, `after_transition()`, `transition_check()`, `on_enter_state()` and `on_leave_state()` decorators;
 - The `Workflow` and `WorkflowEnabled` classes;
 - The `WorkflowError`, `AbortTransition`, `InvalidTransitionError` and `ForbiddenTransition` exceptions.
-

2.2.1 Exceptions

The `xworkflows` module exposes a few specific exceptions:

exception `xworkflows.WorkflowError`

This is the base for all exceptions from the `xworkflows` module.

exception `xworkflows.AbortTransition(WorkflowError)`

This error is raised whenever a transition call fails, either due to state validation or pre-transition checks.

exception `xworkflows.InvalidTransitionError(AbortTransition)`

This exception is raised when trying to perform a transition from an incompatible state.

exception `xworkflows.ForbiddenTransition` (*AbortTransition*)

This exception will be raised when the `check` parameter of the `transition()` decorator returns a non-True value.

2.2.2 States

States may be represented with different objects:

- `base.State` is a basic state (name and title)
- `base.StateWrapper` is an enhanced wrapper around the `State` with enhanced comparison functions.
- `base.StateProperty` is a class-level property-like wrapper around a `State`.

The State class

class `base.State` (*name, title*)

This class describes a state in the most simple manner: with an internal name and a human-readable title.

name

The name of the `State`; used as an internal representation of the state, this should only contain ascii letters and numbers.

title

The title of the `State`; used for display to users.

The StateWrapper class

class `base.StateWrapper` (*state, workflow*)

Intended for use as a `WorkflowEnabled` attribute, this wraps a `State` with knowledge about the related `Workflow`.

Its `__hash__` is computed from the related `name`. It compares equal to:

- Another `StateWrapper` for the same `State`
- Its `State`
- The `name` of its `State`

state

The wrapped `State`

workflow

The `Workflow` to which this `State` belongs.

transitions()

Returns A list of `Transition` with this `State` as source

The StateProperty class

class `base.StateProperty` (*workflow, state_field_name*)

Special property-like object (technically a data descriptor), this class controls access to the current `State` of a `WorkflowEnabled` object.

It performs the following actions:

- Checks that any set value is a valid *State* from the *workflow* (raises `ValueError` otherwise)
- Wraps retrieved values into a *StateWrapper*

workflow

The *Workflow* to which the attribute is related

field_name

The name of the attribute wrapped by this *StateProperty*.

2.2.3 Workflows

A *Workflow* definition is slightly different from the resulting class.

A few class-level declarations will be converted into advanced objects:

- *states* is defined as a list of two-tuples and converted into a *StateList*
- *transitions* is defined as a list of three-tuples and converted into a *TransitionList*
- *initial_state* is defined as the *name* of the initial *State* of the *Workflow* and converted into that *State*

Workflow definition

A *Workflow* definition must inherit from the *Workflow* class, or use the *base.WorkflowMeta* metaclass for proper setup.

Defining states

The list of states should be defined as a list of two-tuples of (name, title):

```
class MyWorkflow(xworkflows.Workflow):
    states = (
        ('initial', "Initial"),
        ('middle', "Intermediary"),
        ('final', "Final - all is said and done."),
    )
```

This is converted into a *StateList* object.

class base.StateList

This class acts as a mixed dictionary/object container of *states*.

It replaces the *states* list from the *Workflow* definition.

__len__()

Returns the number of states in the *Workflow*

__getitem__()

Allows retrieving a *State* from its name or from an instance, in a dict-like manner

__getattr__()

Allows retrieving a *State* from its name, as an attribute of the *StateList*:

```
MyWorkflow.states.initial == MyWorkflow.states['initial']
```

`__iter__()`
Iterates over the states, in the order they were defined

`__contains__()`
Tests whether a *State* instance or its *name* belong to the *Workflow*

Defining transitions

At a *Workflow* level, transition are defined in a list of three-tuples:

- transition name
- list of the *names* of source *states* for the transition, or name of the source state if unique
- *name* of the target *State*

```
class MyWorkflow(xworkflows.Workflow):
    transitions = (
        ('advance', 'initial', 'middle'),
        ('end', ['initial', 'middle'], 'final'),
    )
```

This is converted into a *TransitionList* object.

class `base.TransitionList`

This acts as a mixed dictionary/object container of *transitions*.

It replaces the *transitions* list from the *Workflow* definition.

`__len__()`
Returns the number of transitions in the *Workflow*

`__getitem__()`
Allows retrieving a *Transition* from its name or from an instance, in a dict-like manner

`__getattr__()`
Allows retrieving a *Transition* from its name, as an attribute of the *TransitionList*:

```
MyWorkflow.transitions.accept == MyWorkflow.transitions['accept']
```

`__iter__()`
Iterates over the transitions, in the order they were defined

`__contains__()`
Tests whether a *Transition* instance or its *name* belong to the *Workflow*

`available_from(state)`
Retrieve the list of *Transition* available from the given *State*.

class `base.Transition`

Container for a transition.

name
The name of the *Transition*; should be a valid Python identifier

source
A list of source *states* for this *Transition*

target
The target *State* for this *Transition*

Workflow attributes

A *Workflow* should inherit from the *Workflow* base class, or use the *WorkflowMeta* metaclass (that builds the *states*, *transitions*, *initial_state* attributes).

class xworkflows.**Workflow**

This class holds the definition of a workflow.

states

A *StateList* of all *State* for this *Workflow*

transitions

A *TransitionList* of all *Transition* for this *Workflow*

initial_state

The initial *State* for this *Workflow*

log_transition (*transition*, *from_state*, *instance*, **args*, ***kwargs*)

Parameters

- **transition** – The *Transition* just performed
- **from_state** – The source *State* of the instance (before performing a transition)
- **instance** – The object undergoing a transition
- **args** – All non-keyword arguments passed to the transition implementation
- **kwargs** – All keyword arguments passed to the transition implementation

This method allows logging all transitions performed by objects using a given workflow.

The default implementation logs to the logging module, in the base logger.

implementation_class

The class to use when creating *ImplementationWrapper* for a *WorkflowEnabled* using this *Workflow*.

Defaults to *ImplementationWrapper*.

class base.**WorkflowMeta**

This metaclass will simply convert the *states*, *transitions* and *initial_state* class attributes into the related *StateList*, *TransitionList* and *State* objects.

During this process, some sanity checks are performed:

- Each source/target *State* of a *Transition* must appear in *states*
- The *initial_state* must appear in *states*.

2.2.4 Applying workflows

In order to use a *Workflow*, related objects should inherit from the *WorkflowEnabled* class.

class xworkflows.**WorkflowEnabled**

This class will handle all specific setup related to using *workflows*:

- Converting `attr = SomeWorkflow()` into a *StateProperty* class attribute
- Wrapping all *transition()*-decorated functions into *ImplementationProperty* wrappers
- Adding noop implementations for other transitions

`__add_workflow` (*mcs, field_name, state_field, attrs*)
 Adds a workflow to the attributes dict of the future class.

Parameters

- **`field_name`** (*str*) – Name of the field at which the field holding the current state will live
- **`state_field`** (*StateField*) – The *StateField* as returned by `__find_workflows()`
- **`attrs`** (*dict*) – Attribute dict of the future class, updated with the new *StateProperty*.

Note: This method is also an extension point for custom XWorkflow-related libraries.

`__find_workflows` (*mcs, attrs*)
 Find all workflow definitions in a class attributes dict.

Parameters **`attrs`** (*dict*) – Attribute dict of the future class

Returns A dict mapping a field name to a *StateField* describing parameters for the workflow

Note: This method is also an extension point for custom XWorkflow-related libraries.

`__workflows`
 This class-level attribute holds a dict mapping an attribute to the related *Workflow*.

Note: This is a private attribute, and may change at any time in the future.

`__workflows_implements`
 This class-level attribute holds a dict mapping an attribute to the related implementations.

Note: This is a private attribute, and may change at any time in the future.

class `base.WorkflowEnabledMeta`

This metaclass handles the parsing of *WorkflowEnabled* and related magic.

Most of the work is handled by *ImplementationList*, with one instance handling each *Workflow* attached to the *WorkflowEnabled* object.

2.2.5 Customizing transitions

A bare *WorkflowEnabled* subclass definition will be automatically modified to include “noop” implementations for all transitions from related workflows.

In order to customize this behaviour, one should use the `transition()` decorator on methods that should be called when performing transitions.

`xworkflows.transition` (*[trname=”, field=”, check=None, before=None, after=None]*)
 Decorates a method and uses it for a given *Transition*.

Parameters

- **trname** (*str*) – Name of the transition during which the decorated method should be called. If empty, the name of the decorated method is used.
- **field** (*str*) – Name of the field this transition applies to; useful when two workflows define a transition with the same name.
- **check** (*callable*) – An optional function to call before running the transition, with the about-to-be-modified instance as single argument.
Should return `True` if the transition can proceed.
Deprecated since version 0.4.0: Will be removed in 0.5.0; use `transition_check()` instead.
- **before** (*callable*) – An optional function to call after checks and before the actual implementation.
Receives the same arguments as the transition implementation.
Deprecated since version 0.4.0: Will be removed in 0.5.0; use `before_transition()` instead.
- **after** (*callable*) – An optional function to call *after* the transition was performed and logged.
Receives the instance, the implementation return value and the implementation arguments.
Deprecated since version 0.4.0: Will be removed in 0.5.0; use `after_transition()` instead.

class `base.TransitionWrapper`

Actual class holding all values defined by the `transition()` decorator.

func

The decorated function, wrapped with a few checks and calls.

Hooks

Hooks are declared through a `_HookDeclaration` decorator, which attaches a specific `xworkflows_hook` attribute to the decorated method. Methods with such attribute will be collected into `Hook` objects containing all useful fields.

Registering hooks

`xworkflows._make_hook_dict` (*function*)

Ensures that the given function has a `xworkflows_hook` attributes, and returns it.

The `xworkflows_hook` is a dict mapping each hook kind to a list of (field, hook) pairs:

```
function.xworkflows_hook = {
    HOOK_BEFORE: [('state', <Hook: ...>), ('', <Hook: ...>)],
    HOOK_AFTER: [],
    ...
}
```

Note: Although the `xworkflows_hook` is considered a private API, it may become an official extension point in future releases.

class `base._HookDeclaration`

Base class for hook declaration decorators.

It accepts an (optional) list of transition/state *names*, and *priority* / *field* as keyword arguments:

```
@_HookDeclaration('foo', 'bar')
@_HookDeclaration(priority=42)
@_HookDeclaration('foo', field='state1')
@_HookDeclaration(priority=42, field='state1')
def hook(self):
    pass
```

names

List of *transition* or *state* names the hook applies to

Type str list

priority

The priority of the hook

Type int

field

The name of the StateWrapper field whose transitions the hook applies to

Type str

__as_hook (*self, func*)

Create a Hook for the given callable

__call__ (*self, func*)

Create a Hook for the function, and store it in the function's `xworkflows_hook` attribute.

`xworkflows.before_transition` (**names, priority=0, field=""*)

Marks a method as a pre-transition hook. The hook will be called just before changing a *WorkflowEnabled* object state, with the same **args* and ***kwargs* as the actual implementation.

`xworkflows.transition_check` (**names, priority=0, field=""*)

Marks a method as a transition check hook.

The hook will be called when using `is_available()` and before running the implementation, without any args, and should return a boolean indicating whether the transition may proceed.

`xworkflows.after_transition` (**names, priority=0, field=""*)

Marks a method as a post-transition hook

The hook will be called immediately after the state update, with:

- *res*, return value of the actual implementation
- **args* and ***kwargs* that were passed to the implementation

`xworkflows.on_leave_state` (**names, priority=0, field=""*)

Marks a method as a pre-transition hook to call when the object leaves one of the given states.

The hook will be called with the same arguments as a `before_transition()` hook.

`xworkflows.on_enter_state` (**names, priority=0, field=""*)

Marks a method as a post-transition hook to call just after changing the state to one of the given states.

The hook will be called with the same arguments as a `after_transition()` hook.

Calling hooks

`xworkflows.HOOK_BEFORE`

The kind of `before_transition()` hooks

`xworkflows.HOOK_CHECK`

The kind of `transition_check()` hooks

`xworkflows.HOOK_AFTER`

The kind of `after_transition()` hooks

`xworkflows.HOOK_ON_ENTER`

The kind of `on_leave_state()` hooks

`xworkflows.HOOK_ON_LEAVE`

The kind of `on_enter_state()` hooks

class `base.Hook`

Describes a hook, including its *kind*, *priority* and the list of transitions it applies to.

kind

One of `HOOK_BEFORE`, `HOOK_AFTER`, `HOOK_CHECK`, `HOOK_ON_ENTER` or `HOOK_ON_LEAVE`; the kind of hook.

priority

The priority of the hook, as an integer defaulting to 0. Hooks with higher priority will be executed first; hooks with the same priority will be sorted according to the *function* name.

Type `int`

function

The actual hook function to call. Arguments passed to that function depend on the hook's *kind*.

Type `callable`

names

Name of *states* or *transitions* this hook applies to; will be `('*',)` if the hook applies to all states/transitions.

Type `str tuple`

applies_to (*self*, *transition* [, *from_state*=None])

Check whether the hook applies to the given *Transition* and optional source *State*.

If *from_state* is None, the test means “could the hook apply to the given transition, in at least one source state”.

If *from_state* is not None, the test means “does the hook apply to the given transition for this specific source state”.

Returns `bool`

__call__ (*self*, **args*, ***kwargs*):

Call the hook

__eq__ (*self*, *other*)

__ne__ (*self*, *other*)

Two hooks are “equal” if they wrap the same function, have the same kind, priority and names.

__cmp__ (*self*, *other*)

Hooks are ordered by descending priority and ascending decorated function name.

Advanced customization

Once *WorkflowEnabledMeta* has updated the *WorkflowEnabled* subclass, all transitions – initially defined and automatically added – are replaced with a *base.ImplementationProperty* instance.

class *base.ImplementationProperty*

This class holds all objects required to instantiate a *ImplementationWrapper* whenever the attribute is accessed on an instance.

Internally, it acts as a ‘non-data descriptor’, close to *property()*.

__get__ (*self, instance, owner*)

This method overrides the *getattr()* behavior:

- When called without an instance (*instance=None*), returns itself
- When called with an instance, this will instantiate a *ImplementationWrapper* attached to that instance and return it.

add_hook (*self, hook*)

Register a new *Hook*.

class *base.ImplementationWrapper*

This class handles applying a *Transition* to a *WorkflowEnabled* object.

instance

The *WorkflowEnabled* object to modify when *calling* this wrapper.

field_name

The name of the field modified by this *ImplementationProperty* (a string)

Type str

transition

The *Transition* performed by this object.

Type *Transition*

workflow

The *Workflow* to which this *ImplementationProperty* relates.

Type *Workflow*

implementation

The actual method to call when performing the transition. For undefined implementations, uses *noop()*.

Type callable

hooks

All hooks that may be applied when performing the related transition.

Type dict mapping a hook kind to a list of *Hook*

current_state

Actually a property, retrieve the current state from the instance.

Type *StateWrapper*

__call__ ()

This method allows the *TransitionWrapper* to act as a function, performing the whole range of checks and hooks before and after calling the actual *implementation*.

is_available ()

Determines whether the wrapped transition implementation can be called. In details:

- it makes sure that the current state of the instance is compatible with the transition;
- it calls the `transition_check()` hooks, if defined.

Return type `bool`

`base.noop(instance)`

The ‘do-nothing’ function called as default implementation of transitions.

Collecting the `ImplementationProperty`

Warning: This documents private APIs. Use at your own risk.

Building the list of `ImplementationProperty` for a given `WorkflowEnabled`, and generating the missing ones, is a complex job.

class `base.ImplementationList`

This class performs a few low-level operations on a `WorkflowEnabled` class:

- Collecting `TransitionWrapper` attributes
- Converting them into `ImplementationProperty`
- Adding `noop()` implementations for remaining `Transition`
- Updating the class attributes with those `ImplementationProperty`

state_field

The name of the attribute (from `attr = SomeWorkflow()` definition) currently handled.

Type `str`

workflow

The `Workflow` this `ImplementationList` refers to

implementations

Dict mapping a transition name to the related `ImplementationProperty`

Type `dict(str => ImplementationProperty)`

transitions_at

Dict mapping the name of a transition to the attribute holding its `ImplementationProperty`:

```
@transition('foo')
def bar(self):
    pass
```

will translate into:

```
self.implementations == {'foo': <ImplementationProperty for 'foo' on 'state':
↳<function bar at 0xdeadbeed>>}
self.transitions_at == {'foo': 'bar'}
```

custom_implements

Set of name of implementations which were remapped within the workflow.

load_parent_implements (`self, parent_implements`)

Loads implementations defined in a parent `ImplementationList`.

Parameters `parent_implements` (*ImplementationList*) – The *ImplementationList* from a parent

get_custom_implementations (*self*)
Retrieves definition of custom (non-automatic) implementations from the current list.

Yields (*trname*, *attr*, *implem*): Tuples containing the transition name, the name of the attribute its implementation is stored at, and that implementation (a *ImplementationProperty*).

should_collect (*self*, *value*)
Whether a given attribute value should be collected in the current list.

Checks that it is a *TransitionWrapper*, for a *Transition* of the current *Workflow*, and relates to the current *state_field*.

collect (*self*, *attrs*)
Collects all *TransitionWrapper* from an attribute dict if they verify *should_collect()*.

Raises *ValueError* If two *TransitionWrapper* for a same *Transition* are defined in the attributes.

add_missing_implementations (*self*)
Registers *noop()* *ImplementationProperty* for all *Transition* that weren't collected in the *collect()* step.

register_hooks (*self*, *cls*)
Walks the class attributes and collects hooks from those with a *xworkflows_hook* attribute (through *register_function_hooks()*)

register_function_hooks (*self*, *func*)
Retrieves hook definitions from the given function, and registers them on the related *ImplementationProperty*.

_may_override (*self*, *implem*, *other*)
Checks whether the *implem ImplementationProperty* is a valid override for the other *ImplementationProperty*.

Rules are:

- A *ImplementationProperty* may not override another *ImplementationProperty* for another *Transition* or another *state_field*
- A *ImplementationProperty* may not override a *TransitionWrapper* unless it was generated from that *TransitionWrapper*
- A *ImplementationProperty* may not override other types of previous definitions.

fill_attrs (*self*, *attrs*)
Adds all *ImplementationProperty* from *implementations* to the given attributes dict, unless *_may_override()* prevents the operation.

transform (*self*, *attrs*)

Parameters `attrs` (*dict*) – Mapping holding attribute declarations from a class definition

Performs the following actions, in order:

- *collect()*: Create *ImplementationProperty* from the *transition wrappers* in the *attrs* dict
- *add_missing_implementations()*: create *ImplementationProperty* for the remaining *transitions*

- `fill_attrs()`: Update the `attrs` dict with the *implementations* defined in the previous steps.

2.3 ChangeLog

2.3.1 1.1.0 (2021-04-29)

New:

- Add support for Python 3.7, 3.8, 3.9

2.3.2 1.0.4 (2014-08-11)

Bugfix:

- Fix `setup.py` execution on Python3 or non-UTF locale.

2.3.3 1.0.3 (2014-01-29)

Bugfix:

- Allow setting the current state of a `WorkflowEnabled` instance from a state's name
- Ensure `states` behaves as a proper mapping

2.3.4 1.0.2 (2013-09-24)

Bugfix:

- Fix installation from PyPI

2.3.5 1.0.1 (2013-09-24)

Misc:

- Switch back to `setuptools` `>= 0.8` for packaging.

2.3.6 1.0.0 (2013-04-29)

Bugfix:

- Fix hook registration on custom implementations while inheriting *WorkflowEnabled*.

New:

- Add support for Python 2.6 to 3.2

Backward incompatible:

- The string representation of *State* and *StateWrapper* now reflects the state's name, as does their `unicode()` representation in Python 2.X.

2.3.7 0.4.1 (2012-08-03)

Bugfix:

- Support passing a *Transition* or a *State* to hooks, instead of its name.

2.3.8 0.4.0 (2012-08-02)

New:

- Improve support for transition hooks, with the `xworkflows.before_transition()`, `xworkflows.after_transition()`, `xworkflows.transition_check()`, `xworkflows.on_enter_state()` and `xworkflows.on_leave_state()` decorators.

Bugfix:

- Fix support for inheritance of `xworkflows.WorkflowEnabled` objects.

Deprecated:

- Use of the `check=`, `before=`, `after=` keyword arguments in the `@transition` decorator is now deprecated; use `@before_transition`, `@after_transition` and `@transition_check` instead. Support for old keyword arguments will be removed in 0.5.0.

Backward incompatible:

- The (private) `ImplementationWrapper` class no longer accepts the `check`, `before`, `after` arguments (use `hooks` instead)

2.3.9 0.3.2 (2012-06-05)

Bugfix:

- Fix transition logging for objects whose `__repr__` doesn't convert to unicode.

2.3.10 0.3.1 (2012-05-29)

Bugfix:

- Make the `title` argument mandatory in *State* initialization

2.3.11 0.3.0 (2012-04-30)

New:

- Allow and document customization of the `ImplementationWrapper`
- Add a method to check whether a transition is available from the current instance
- Cleanup `ImplementationList` and improve its documentation

2.3.12 0.2.4 (23 04 2012)

New:

- Improve documentation
- Add pre-transition `check` hook
- Remove alternate *Workflow* definition schemes.
- Properly validate objects using two workflows with conflicting transitions.

2.3.13 0.2.3 (15 04 2012)

New:

- Simplify API
- Add support for pre/post transition and logging hooks

2.3.14 0.2.1 (26 03 2012)

New:

- Add support for workflow subclassing
- Improve packaging

2.3.15 0.1.0 (08 09 2011)

New:

- First Public Release.

CHAPTER 3

Resources

- Package on PyPI: <http://pypi.python.org/pypi/xworkflows>
- Repository and issues on GitHub: <http://github.com/rbarrois/xworkflows>
- Doc on <http://readthedocs.org/docs/xworkflows/>

CHAPTER 4

Indices and tables

- `genindex`
- `modindex`
- `search`

X

xworkflows, [14](#)

Symbols

`__call__()` (*xworkflows.base.ImplementationWrapper* method), 23
`__call__()` (*xworkflows.base._HookDeclaration* method), 21
`__cmp__()` (*xworkflows.base.Hook* method), 22
`__contains__()` (*xworkflows.base.StateList* method), 17
`__contains__()` (*xworkflows.base.TransitionList* method), 17
`__eq__()` (*xworkflows.base.Hook* method), 22
`__get__()` (*xworkflows.base.ImplementationProperty* method), 23
`__getattr__()` (*xworkflows.base.StateList* method), 16
`__getattr__()` (*xworkflows.base.TransitionList* method), 17
`__getitem__()` (*xworkflows.base.StateList* method), 16
`__getitem__()` (*xworkflows.base.TransitionList* method), 17
`__iter__()` (*xworkflows.base.StateList* method), 16
`__iter__()` (*xworkflows.base.TransitionList* method), 17
`__len__()` (*xworkflows.base.StateList* method), 16
`__len__()` (*xworkflows.base.TransitionList* method), 17
`__ne__()` (*xworkflows.base.Hook* method), 22
`_add_workflow()` (*xworkflows.WorkflowEnabled* method), 18
`_as_hook()` (*xworkflows.base._HookDeclaration* method), 21
`_find_workflows()` (*xworkflows.WorkflowEnabled* method), 19
`_make_hook_dict()` (*in module xworkflows*), 20
`_may_override()` (*xworkflows.base.ImplementationList* method), 25

`_workflows` (*xworkflows.WorkflowEnabled* attribute), 19
`_xworkflows_implements` (*xworkflows.WorkflowEnabled* attribute), 19

A

`AbortTransition`, 14
`add_hook()` (*xworkflows.base.ImplementationProperty* method), 23
`add_missing_implementations()` (*xworkflows.base.ImplementationList* method), 25
`after_transition()` (*in module xworkflows*), 21
`applies_to()` (*xworkflows.base.Hook* method), 22
`available_from()` (*xworkflows.base.TransitionList* method), 17

B

`base._HookDeclaration` (*class in xworkflows*), 20
`base.Hook` (*class in xworkflows*), 22
`base.ImplementationList` (*class in xworkflows*), 24
`base.ImplementationProperty` (*class in xworkflows*), 23
`base.ImplementationWrapper` (*class in xworkflows*), 23
`base.noop()` (*in module xworkflows*), 24
`base.State` (*class in xworkflows*), 15
`base.StateList` (*class in xworkflows*), 16
`base.StateProperty` (*class in xworkflows*), 15
`base.StateWrapper` (*class in xworkflows*), 15
`base.Transition` (*class in xworkflows*), 17
`base.TransitionList` (*class in xworkflows*), 17
`base.TransitionWrapper` (*class in xworkflows*), 20
`base.WorkflowEnabledMeta` (*class in xworkflows*), 19
`base.WorkflowMeta` (*class in xworkflows*), 18
`before_transition()` (*in module xworkflows*), 21

C

`collect()` (*xworkflows.base.ImplementationList method*), 25

`current_state` (*xworkflows.base.ImplementationWrapper attribute*), 23

`custom_implements` (*xworkflows.base.ImplementationList attribute*), 24

F

`field` (*xworkflows.base._HookDeclaration attribute*), 21

`field_name` (*xworkflows.base.ImplementationWrapper attribute*), 23

`field_name` (*xworkflows.base.StateProperty attribute*), 16

`fill_attrs()` (*xworkflows.base.ImplementationList method*), 25

`ForbiddenTransition`, 14

`func` (*xworkflows.base.TransitionWrapper attribute*), 20

`function` (*xworkflows.base.Hook attribute*), 22

G

`get_custom_implementations()` (*xworkflows.base.ImplementationList method*), 25

H

`HOOK_AFTER` (*in module xworkflows*), 22

`HOOK_BEFORE` (*in module xworkflows*), 22

`HOOK_CHECK` (*in module xworkflows*), 22

`HOOK_ON_ENTER` (*in module xworkflows*), 22

`HOOK_ON_LEAVE` (*in module xworkflows*), 22

`hooks` (*xworkflows.base.ImplementationWrapper attribute*), 23

I

`implementation` (*xworkflows.base.ImplementationWrapper attribute*), 23

`implementation_class` (*xworkflows.Workflow attribute*), 18

`implementations` (*xworkflows.base.ImplementationList attribute*), 24

`initial_state` (*xworkflows.Workflow attribute*), 18

`instance` (*xworkflows.base.ImplementationWrapper attribute*), 23

`InvalidTransitionError`, 14

`is_available()` (*xworkflows.base.ImplementationWrapper method*), 23

K

`kind` (*xworkflows.base.Hook attribute*), 22

L

`load_parent_implements()` (*xworkflows.base.ImplementationList method*), 24

`log_transition()` (*xworkflows.Workflow method*), 18

N

`name` (*xworkflows.base.State attribute*), 15

`name` (*xworkflows.base.Transition attribute*), 17

`names` (*xworkflows.base._HookDeclaration attribute*), 21

`names` (*xworkflows.base.Hook attribute*), 22

O

`on_enter_state()` (*in module xworkflows*), 21

`on_leave_state()` (*in module xworkflows*), 21

P

`priority` (*xworkflows.base._HookDeclaration attribute*), 21

`priority` (*xworkflows.base.Hook attribute*), 22

R

`register_function_hooks()` (*xworkflows.base.ImplementationList method*), 25

`register_hooks()` (*xworkflows.base.ImplementationList method*), 25

S

`should_collect()` (*xworkflows.base.ImplementationList method*), 25

`source` (*xworkflows.base.Transition attribute*), 17

`state` (*xworkflows.base.StateWrapper attribute*), 15

`state_field` (*xworkflows.base.ImplementationList attribute*), 24

`states` (*xworkflows.Workflow attribute*), 18

T

`target` (*xworkflows.base.Transition attribute*), 17

`title` (*xworkflows.base.State attribute*), 15

`transform()` (*xworkflows.base.ImplementationList method*), 25

`transition` (*xworkflows.base.ImplementationWrapper attribute*), 23

`transition()` (*in module xworkflows*), 19

`transition_check()` (*in module xworkflows*), 21
`transitions` (*xworkflows.Workflow attribute*), 18
`transitions()` (*xworkflows.base.StateWrapper method*), 15
`transitions_at` (*xworkflows.base.ImplementationList attribute*), 24

W

`Workflow` (*class in xworkflows*), 18
`workflow` (*xworkflows.base.ImplementationList attribute*), 24
`workflow` (*xworkflows.base.ImplementationWrapper attribute*), 23
`workflow` (*xworkflows.base.StateProperty attribute*), 16
`workflow` (*xworkflows.base.StateWrapper attribute*), 15
`WorkflowEnabled` (*class in xworkflows*), 18
`WorkflowError`, 14

X

`xworkflows` (*module*), 14